Git Revert & Git Reset

Both git revert and git reset are powerful tools for undoing changes in Git, but they work very differently and serve distinct purposes. This guide explains their differences, use cases, and detailed examples.

Overview

Feature	Git Revert	Git Reset
Purpose	Undo changes by creating a new commit.	Undo changes by moving the branch pointer.
History	Preserves commit history.	Rewrites commit history (in some cases).
Use Case	Safe for shared/public branches.	Typically used on local/private branches.
Impact	Creates a new commit to reverse changes.	Modifies the working directory and staging area.
Modes	No modes (always creates a new commit).	--soft, --mixed, --hard modes.

Git Revert

git revert creates a new commit that reverses the changes introduced by a specific commit. It does not remove the original commit from the history, making it **safe to use on shared branches**.

Syntax

```
git revert <commit-hash>
```

Example 1: Reverting a Single Commit

Scenario:

You have the following commit history:

A --- B --- C --- D (main)

Commit D introduces a bug, and you want to undo its changes.

Commands:

1. Identify the commit hash for D:

```
git log --oneline
```

Output:

d4e5f1a D: Introduce bug
c2b4a8b C: Add new feature
a7c3d9e B: Update README
9a8f7g6 A: Initial commit

2. Revert commit D:

```
git revert d4e5f1a
```

3. Resulting History:

A --- B --- C --- D --- E (main)

\

E: Revert "Introduce bug"

- Commit E is a new commit that reverses the changes introduced by D.

Example 2: Reverting Multiple Commits

Scenario:

You want to undo commits C and D.

Commands:

1. Revert commits one by one:

```
git revert d4e5f1a
git revert c2b4a8b
```

2. Resulting History:

A --- B --- C --- D --- E --- F (main)

\

E: Revert "Introduce bug"

F: Revert "Add new feature"

Example 3: Reverting with Conflict

Scenario:

If reverting a commit conflicts with changes in the current branch, Git pauses the process.

Commands:

1. Revert a commit that conflicts:

```
git revert <conflicting-commit-hash>
```

2. Git reports a conflict:

CONFLICT (content): Merge conflict in file.txt

3. Resolve the conflict, then continue:

```
git add <resolved-file>
git revert --continue
```

Git Reset

git reset moves the branch pointer backward to a specific commit, effectively "resetting" the branch to a previous state. It has three modes that control how it impacts the working directory and staging area.

Modes of Git Reset

Mode	Effect on Working Directory	Effect on Staging Area	Use Case
--soft	No change	Retains changes	Undo commits while keeping changes staged.
--mixed	No change	Clears staging area	Undo commits and unstage changes (default mode).
--hard	Discards changes	Clears staging area	Completely undo commits and changes.

Syntax

```
git reset [<mode>] <commit-hash>
```

If no mode is specified, --mixed is used by default.

Example 1: Reset to a Previous Commit (Default --mixed)

Scenario:

You have the following commit history:

A --- B --- C --- D (main)

You want to reset the branch to commit B, removing C and D from the history.

Commands:

1. Reset to B:

```
git reset B
```

2. Resulting State:

A --- B (main)

\

C --- D (unstaged changes)

- The changes from C and D are moved to the working directory as unstaged changes.

Example 2: Soft Reset

Scenario:

You want to undo the last commit but keep the changes staged.

Commands:

1. Reset to the previous commit:

```
git reset --soft HEAD~1
```

2. Result:

- The last commit is removed, but the changes remain staged.

Example 3: Hard Reset

Scenario:

You want to completely discard the last commit and its changes.

Commands:

1. Reset to the previous commit:

```
git reset --hard HEAD~1
```

2. Result:

A --- B --- C (main)

- The commit and its changes are permanently removed.

Example 4: Reset with Unstaged Changes

Scenario:

You have staged changes that you want to unstage and keep in the working directory.

Commands:

1. Reset the staging area:

```
git reset --mixed
```

Git Revert vs. Git Reset

Feature	Git Revert	Git Reset
History Preservation	Preserves history with a new commit.	Rewrites history by moving the branch pointer.
Usage	Safe for public/shared branches.	Best for private/local branches.
Impact on Files	Does not impact the working directory.	Depends on the reset mode (soft, mixed, or hard).
Undo Specific Commits	Creates a new commit to reverse specific commits.	Removes commits entirely (rewrites history).

When to Use Git Revert or Git Reset

Use Git Revert When:

- You need to undo changes in a shared/public branch.
- You want to preserve the original commit history.

Use Git Reset When:

- You are working on a local branch and want to rewrite history.
- You need to undo commits and clean up your working directory.

Common Issues and Best Practices

Avoid Losing Data with Reset

- Use git reset --soft or --mixed if unsure about discarding changes.
- Avoid --hard unless you're confident you don't need the changes.

Avoid Overusing Revert

- For large or multiple commits, reverting one by one can clutter history. Use git reset for simpler workflows.

Summary

- git revert is the safer option for undoing changes, as it creates a new commit to reverse changes and preserves history.
- git reset is more flexible but can be destructive, especially in --hard mode.